TEACHING ASSURANCE USING CHECKLISTS

Keynote

Matt Bishop

University of California, Davis

Abstract: Industry, the government, and many other entities have expressed concern about the way schools are teaching programming. The central theme is that students do not learn how to program "securely." Several proposals to improve the quality of programming involve checklists, or items that that students must demonstrate mastery of in order to achieve the appropriate goal (such as passing the class or graduating).

Underlying these proposals is the goal of teaching *assurance*—building programs and systems that meet specific requirements—and making it a part of everyday work, as opposed to a specialized discipline applied only when there is time, or when there are specific assurance-related requirements. This raises several issues, especially when one is considering "security assurance."

What is "security?" The standard answer, "security is whatever the security policy defines it to be," does not provide the guidance needed to develop a new student's intuition about how to develop programs. Instead, one needs to focus on a specific set of security requirements. No single set covers all situations, but certain elements arise in many requirements. For example, the requirement that a program handle error conditions appropriately means that a student must write programs in which error conditions are detected. Buffer overflows may not cause security breaches (especially if availability is not at issue), but they do cause errors to arise. Thus, a better pedagogic question than "how can we teach students to write more secure programs" is "what principles should we be teaching students to enable them to write more secure programs, and how can we teach them how to apply these principles to specific situations?"

This formulation recognizes that the term "secure programming" is imprecise because it means different things to different people, and in different environments. It also adds to our burden because we must teach students to evaluate situations and determine what "security" means *in the specific case*. This requires teaching principles.

History plays a part in this. Early papers, such as the Anderson Report, described threats to systems and created a framework for addressing them. These are important not only because current technology implements many of the ideas, but also because the framework itself teaches one how to think about assurance. Consider the reference monitor, a "guardian at the gate" that controls access to a resource. This means that *all* accesses of the resource must pass through the reference monitor. Spaghetti code, which allows multiple paths of access without a rigorous examination of *why* each path of access must be present, leads to poor coding and (potentially) security problems. The reference monitor must be checked, to ensure it works correctly; this leads to the requirement of smallness, so it can be validated. Complexity may be required, but unnecessary complexity is a weakness. Striving for simplicity, of style if not of content, is the mark of a good programmer. Finally, if an attacker can alter the reference monitor, or the data upon which it relies, that subverted control will allow unauthorized access to the resource. Hence the programmer must guard against such tampering, by (for example) checking error conditions to prevent attackers from exploiting them. From one framework comes several ideas central to protection.

Over time, technologies change, and methodologies adapt to new circumstances. But principles do not change. They provide guidance for developing new methodologies and technologies. Because of the rapid pace at which our field, computer science, is evolving, focusing on principles and teaching students how to analyze situations, and apply these principles, is critical.

Checklists have a part to play in this process. There are many different types of checklists. Such lists can provide guidance or specific items to consider, and may be used by a student or a grader (auditor).

– A checklist can simply list items to prompt students' memory, for example "check for possible errors and handle them." The usefulness of this type of checklist depends entirely on the student's ability to translate the items on the checklist into the particular domain in which the student is working. This list provides guidance, and the student is expected to look beyond the list as appropriate.

– A checklist can list specific items that students are required to satisfy. These checklists are usually specific, for example "check the return values of all functions to ensure the function worked properly." These checklists are useful for reminding the student about details, but risk the student performing the items on the checklist without considering their appropriateness, and not looking beyond the checklist for items that need to be considered.

– A checklist can be used to aid in the evaluation of a student's work, or of the result of that work. Again, this checklist is primarily a guide to the grader or auditor, who is expected to translate the generality of the items into specific criteria appropriate for the work.

– A checklist can list specific items that the student's work is to satisfy. Here, the checklist requires the assessor to determine if the item is met. If so, it is checked off; if not, the item is marked unsatisfied. The set of satisfied (or unsatisfied) items determines the score.

The best checklists are derived from principles, and their items develop logically through the derivation of principles, methodology, and application to a particular domain, guided by experience of practitioners. This allows one to justify the checklist rigorously and to see how the principles strengthen the practice; assurance at its best.

The type of education being sought, and the type of environment in which the checklist is used, determines the appropriateness of any particular checklist. Used properly, a checklist can enhance a student's education; used improperly, that same checklist can hinder the student's progress, as well as fail to achieve the goals of that student's education.

ACKNOWLEDGEMENT

This paper expands upon ideas discussed in a column co-authored with Deborah Frincke [2]. I am grateful to her for the discussions that helped me sharpen some of these ideas, and for her encouragement.

REFERENCES

1. J. Anderson, *Computer Security Technology Planning Study*, Technical Report ESD-TR-73-51, Electronic Systems Division, Hanscom Air Force Base, Hanscom, MA (1974).
2. M. Bishop and D. Frincke, "Teaching Secure Programming," *IEEE Security and Privacy* **3**(5) pp. 54–56 (Sep. 2005).